UNCLASSIFIED

(2)

Date Entered

AD-A211 679

ATION PAGE

12. GOVT ACCESSION NO	3. RECIPIENT'S CATALOG NUMBER
Ada Compiler Validation Summary Report: Concurrent Computer Corporation, C^3 Ada, Version R02-02.00, Concurrent Computer Corporation 3280 MPS (Host & Target), 890711W1.10178	5. TYPE OF REPORT & PERIOD COVERED 11 July 1989 to 11 July 1990 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Concurrent Computer Corporation, C^3 Ada, Version R02-02.00, Wright-Patterson AFB, Concurrent Computer Corporation 3280 MPS under OS/32, Version R08-02.03 (Host & Target), ACVC 1.10.

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
3 JAN 73 S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AVF Control Number: AVF-VSR-289.0789
89-04-11-CCC

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 890711W1.10109
Concurrent Computer Corporation
C^3Ada, Version R02-02.00
Concurrent Computer Corporation 3280 MPS

Completion of On-Site Testing:
11 July 1989

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: C^3 Ada, Version R02-02.00

Certificate Number: 890711W1.10109

Host: Concurrent Computer Corporation 3280 MPS under OS/32, Version R08-02.03

Target: Concurrent Computer Corporation 3280 MPS under OS/32, Version R08-02.03

Testing Completed 11 July 1989 Using ACVC 1.10

This report has been reviewed and is approved.

Steve P. Wilson

Ada Validation Facility
Steve P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

for Edward N. Liebhardt

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

Deputy Director

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 11 July 1989 at Tinton Falls NJ.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including

cross-compilers, translators, and interpreters.

Failed test — An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host — The computer on which the compiler resides.

Inapplicable test — An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test — An ACVC test for which a compiler generates the expected result.

Target — The computer for which a compiler generates code.

Test — A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test — An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every

illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate

tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: C^{3}Ada, Version R02-02.00

ACVC Version: 1.10

Certificate Number: 890711W1.10109

Host Computer:

Machine:	Concurrent Computer Corporation 3280 MPS
Operating System:	OS/32 Version R08-02.03
Memory Size:	16 Megabytes

Target Computer:

Machine:	Concurrent Computer Corporation 3280 MPS
Operating System:	OS/32 Version R08-0[illegible].03
Memory Size:	16 Megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types TINY_INTEGER, SHORT_INTEGER, and LONG_FLOAT in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is not gradual. (See tests C45524A..Z.)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z.)

(2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z.)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to a null array type with INTEGER'LAST + 2 components. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to a null array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array type is declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B, EA3004C..D, and CA3004E..F.)

i. Generics

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

(2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N, and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J, CE2102R, CE2102T, and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K.)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G, CE3104C, CE3110A, and CE3114A.)

(9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

(10) Temporary sequential files are not given names. (See test CE2108A.)

(11) Temporary direct files are not given names. (See test CE2108C.)

(12) Temporary text files are not given names. (See test CE3112A.)

(13) More than one internal file can be associated with each

external file for sequential files when reading or writing, except when the external file is a temporary file. (See tests CE2107A..E, CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when reading or writing, except when the external file is a temporary file. (See tests CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when writing or reading. (See tests CE3111A..E, CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 360 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation . Modifications to the code, processing, or grading for 8 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	128	1132	1963	17	27	46	3313
Inapplicable	1	6	352	0	1	0	360
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	190	576	545	245	172	99	160	331	137	36	252	280	290	3313
Inappl	22	73	135	3	0	0	6	1	0	0	0	89	31	360
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 360 tests were inapplicable for the reasons indicated:

a. C24113D..K (8 tests) are not applicable because this implementation does not support a line length greater than 80 characters.

b. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y	C35705L..Y	C35706L..Y	C35707L..Y
C35708L..Y	C35802L..Z	C45241L..Y	C45321L..Y
C45421L..Y	C45521L..Z	C45524L..Z	C45621L..Z
C45641L..Y	C46012L..Z		

c. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

d. The following 13 tests are not applicable because this implementation does not support 'STORAGE_SIZE representation clauses for task types:

A39005D	C87B62D	CD1009K	CD1009T	CD1009U
CD1C03E	CD1C04B	CD1C06A	CD2C11A	CD2C11B
CD2C11C	CD2C11D	CD2C11E		

e. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

f. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 47.

g. C86001F is not applicable because this implementation does not support recompilation of package SYSTEM.

h. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

i. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

j. C96005B is not applicable because there are no values of type DURATION'BASE that are outside the range of DURATION.

k. CD1009C, CD2A41A..B (2 tests), CD2A41E, and CD2A42A..J (10 tests) are not applicable because this implementation does not support size clauses for floating point types when the size specification does not equal 32.

l. CD2A61A..D (4 tests), CD2A61F, CD2A61H..L (5 tests), and CD2A62A..C (3 tests) are not applicable because this implementation does not support size clauses for array types when the specification is not the default value chosen by the compiler.

m. CD2A71A..D (4 tests) and CD2A72A..D (4 tests) are not applicable because this implementation does not support size clauses for record types when the specification is not the default value

chosen by the compiler.

n. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because this implementation does not support size clauses for access types when the size specification does not equal 32.

o. The following 29 tests are not applicable because this implementation does not support address clauses for initialized constant objects:

CD5011B	CD5011D	CD5011F	CD5011H	CD5011L
CD5011N	CD5011R	CD5012C	CD5012D	CD5012G
CD5012H	CD5012L	CD5013B	CD5013D	CD5013F
CD5013H	CD5013L	CD5013N	CD5013R	CD5014B
CD5014D	CD5014F	CD5014H	CD5014J	CD5014L
CD5014N	CD5014R	CD5014U	CD5014W	

p. CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support address clauses for task units.

q. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

r. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

s. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

t. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

u. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

v. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

w. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

x. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

y. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

z. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

aa. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

ab. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ac. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ad. CE2102V is inapplicable because this implementation supports open with OUT_FILE mode for DIRECT_IO.

ae. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

af. CE2107C..D (2 tests), CE2107H, CE2107L, CE2108B, CE2108D, and CE3112B are not applicable because this implementation does not support temporary files with names.

ag. EE2401D is inapplicable because this implementation does not support DIRECT_IO with unconstrained array types. This implementation raises USE_ERROR on CREATE.

ah. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

ai. CE3102F is inapplicable because this implementation supports RESET for text files.

aj. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

ak. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

al. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

am. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

an. CE3111B and CE3115B are both not applicable because this implementation requires that TEXT_IO.PUT does not write to an external file until a subsequent NEW_LINE, RESET, or CLOSE operation is done. (See Section 3.6)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 8 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

BC2001D BC2001E BC3204B BC3205B BC3205D

The following modifications were made to compensate for legitimate implementation behavior:

a. At the recommendation of the AVO, the expression "2**T'MANTISSA - 1" on line 262 of test CC1223A was changed to "(2**(T'MANTISSA-1)-1 + 2**(T'MANTISSA-1))" since the previous expression causes an unexpected exception to be raised.

The following tests were graded using a modified evaluation criteria:

a. CE3111B and CE3115B both raised an unhandled END_ERROR exception and failed during execution. For this implementation, TEXT_IO.PUT does not write to an external file until a subsequent NEW_LINE, RESET, or CLOSE operation. A GET operation before the NEW_LINE can potentially result in an END_ERROR exception, and this occurs at line 29 in CE3111B and at line 101 in CE3115B. The AVO ruled these tests should be graded as not applicable.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the C^3 Ada was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the C^3 Ada using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	Concurrent Computer Corporation 3280 MPS
Host operating system:	OS/32, Version R08-02.03
Target computer:	Concurrent Computer Corporation 3280 MPS
Target operating system:	OS/32, Version R08-02.03
Compiler:	C^3Ada, Version R02-02.00

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Concurrent Computer Corporation 3280 MPS. Results were printed from the host computer.

The compiler was tested using command scripts provided by Concurrent Computer Corporation and reviewed by the validation team. The compiler was tested using all the following option settings. See Appendix E for a complete listing of the compiler options for this implementation. The following list of compiler options includes those options which were invoked by default:

TEST INFORMATION

```
ABORT => OFF
ALIST => OFF
INFORM => ON
INLINE => ON
LIST => ON
OPTIMIZE => ON
PAGE_SIZE => 60
SEGMENTED => ON
STACK_CHECK => ON
SUMMARY => ON
SUPPRESS_ALL => OFF
SUPPRESS_OVERFLOW => OFF
WARN => ON
```

Tests were compiled, linked, and executed (as appropriate) using a single computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Tinton Falls NJ and was completed on 11 July 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Concurrent Computer Corporation has submitted the following Declaration of Conformance concerning the C^3 Ada.

DECLARATION OF CONFORMANCE

Compiler Implementor: Concurrent Computer Corporation
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

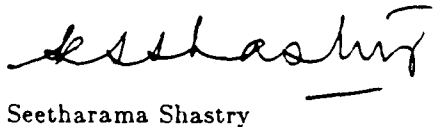
Base Configuration

Base Compiler Name: C ^{3}Ada **Version:** R02-02.00

Base Host Architecture ISA: Concurrent Computer Corporation 3280 MPS
(Under OS/32, Version R08-02.03)
Base Target Architecture ISA: Concurrent Computer Corporation 3280 MPS
(Under OS/32, Version R08-02.03)

Implementor's Declaration

I, the Undersigned, representing Concurrent Computer Corporation have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compilers listed in this declaration. I declare that Concurrent Computer Corporation is the owner of record of the Ada language compilers listed above and, as such, is responsible for maintaining the said compilers in conformance to ANIS/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

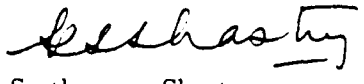


5/3/89

Seetharama Shastry
Senior Manager, System Software Development (Date)

Owner's Declaration

I, the undersigned, representing Concurrent Computer Corporation take full responsibility for implementation and maintenance of the Ada Compilers listed above, and agree to public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



5/3/89

Seetharama Shastry
Senior Manager, System Software Development (Date)

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the C^3 Ada, Version R02-02.00, as described in this Appendix, are provided by Concurrent Computer Corporation. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...

   type INTEGER is range -2147483648 .. 2147483647;
   type SHORT_INTEGER is range -32768 .. 32767;
   type TINY_INTEGER is range -128 .. 127;

   type FLOAT is digits 6 range -16#0.FFFF_FF#E63 .. 16#0.FFFF_FF#E63;
   type LONG_FLOAT is digits 15
        range -16#0.FFFF_FFFF_FFFF_FF#E63 .. 16#0.FFFF_FFFF_FFFF_FF#E63;

   type DURATION is delta 0.00006103515625
        range -131072.00 .. 131071.99993896484375;

   ...

end STANDARD;
```

APPENDIX F
IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.1 INTRODUCTION

The following sections provide all implementation-dependent characteristics of the C [3]Ada Compiler.

F.2 IMPLEMENTATION-DEPENDENT PRAGMAS

The following is the syntax representation of a pragma:

pragma *IDENTIFIER* [(*ARGUMENT* [,*ARGUMENT*)]];

Where:

IDENTIFIER	is the name of the pragma.
ARGUMENT	defines a parameter of the pragma. For example, the LIST pragma expects the arguments ON or OFF.

Table F-1 summarizes all of the recognized pragmas and whether they are implemented or not.

TABLE F-1. SUMMARY OF RECOGNIZED PRAGMAS

PRAGMA	IMPLEMENTED	COMMENTS
BIT_PACK	Yes	This pragma allows packing of composite non-FLOAT type objects to the bit level, thereby achieving greater data compaction. Use of this pragma will result in longer compile and run times.
BYTE_PACK	Yes	The elements of an array or record are packed down to a minimal number of bytes.
CONTROLLED	No	Automatic storage reclamation of unreferenced access objects is not applicable to the C[3]Ada implementation.
ELABORATE	Yes	Is handled as defined by the Ada language.
INLINE	Yes	Subprogram bodies are expanded inline at each call.
INTERFACE	Yes	Is implemented for ASSEMBLER and FORTRAN.
LIST	Yes	Is handled as defined by the Ada language.
MEMORY_SIZE	No	The user cannot specify the number of available storage units in the machine configuration which is defined in package SYSTEM.
OPTIMIZE	No	The user cannot specify either time or space as the primary optimization criterion.
PACK	Yes	The elements of an array or record are packed down to a minimal number of bits.
PAGE	Yes	Is handled as defined by the Ada language.
PARTIAL_IMAGE	Yes	This pragma informs the compiler that the named package may be used to build a partial image, and causes the compiler to verify that the package meets all requirements for such use.
PRIORITY	No	The task or main_program cannot have priority.
SHARED	No	Not applicable because every read or update of the variable declared by an object declaration and whose type is a scalar or access type is a synchronization point for that variable.

TABLE F-1. SUMMARY OF RECOGNIZED PRAGMAS (Continued)

PRAGMA	IMPLEMENTED	COMMENTS
STACK_CHECK	Yes	When specified with the argument OFF, this pragma indicates to the compiler that there is enough space in the initial stack chunk for the activation record of all subroutines that may be active at any time. Therefore, no code is generated which checks for providing additional space for the run-time stack of any task or of the main task. This pragma may appear wherever the use of a pragma is legal.
STORAGE_UNIT	No	The user cannot specify the number of bits per storage unit, which is defined in package SYSTEM.
SUPPRESS	No	All run-time checks, such as ACCESS_CHECK, INDEX_CHECK, RANGE_CHECK, etc., cannot be suppressed for any specific type, object, subprogram etc., See the description of SUPPRESS_ALL.
SUPPRESS_ALL	Yes	This pragma gives the compiler permission to omit all of the following run-time checks for all types and objects in the designated compilation units: ACCESS_CHECK, RANGE_CHECK, LENGTH_CHECK, INDEX_CHECK, DISCRIMINANT_CHECK and OVERFLOW_CHECK for all integer and fixed point calculations. The pragma must be placed before each compilation unit.
SYSTEM_NAME	No	The user cannot specify the target system name, which is defined in package SYSTEM.

F.2.1 Pragma INLINE Restrictions

Inline expansion of a subprogram call will not occur if the following conditions are not satisfied:

1. If the subprogram body is contained in the same compilation unit as the call, the complete text of the body must precede the call. If the subprogram body is not contained in the same compilation unit as the call, the compilation unit containing the body must be compiled before the unit containing the call. (Care should be taken that the unit containing the body is not recompiled, since this would make the unit containing the call obsolete.) For every call of a subprogram for which **pragma** INLINE is given, a warning message is reported if the subprogram body is not already known to the compiler as indicated above; the warning message indicates that inline expansion is not done for that particular call.
2. The subprogram body **and** any enclosed declare blocks may not contain:

   - declarations of subprograms
   - declarations of task types or single tasks
   - body stubs
   - generic instaniations

   For every call of a subprogram for which **pragma** INLINE is given, a warning message is reported if this set of conditions is not satisfied; the message indicates that inline expansion is not done for that particular call.
3. The subprogram body, **excluding** any enclosed declare blocks, may not contain

   - declarations of objects with task subcomponents
   - declarations of access types where the designated type has task subcomponents
   - exception handlers

   For every call of a subprogram for which **pragma** INLINE is given, a warning message is reported if this set of conditions is not satisfied; the message indicates that inline expansion is not done for that particular call.
4. Inline expansion occurs when the expanded code contains a valid subprogram call. However, a duplicate inline expansion is not carried out for a subprogram call if inline

expansion for that subprogram is already in process (e.g., a recursive call). A warning message is generated informing the user that this is the case.

F.3 LENGTH CLAUSES

A length clause specifies the amount of storage associated with a given type. The following is a list of the implementation-dependent attributes.

T'SIZE	Must be 32 for a type derived from FLOAT, and 64 for a type derived from LONG_FLOAT. For array and record types, only the size chosen by the compiler may be specified.
T'STORAGE_SIZE	is fully supported for collection size specification.
T'STORAGE_SIZE	is not supported for task activation. Task memory is limited by the work space for the program.
T'SMALL	must be a power of two for a fixed point type.

Size representation only applies to types - not to subtypes. In the following example, the size of T is 32, but the size of T1 is not necessarily 32.

```
type T is integer range  0..100;
subtype T1 is T range 0..10;
for T'SIZE use 32;
```

In the following example, the size of the subtype is the same as the size of the type (size of the type is applied to the subtype).

```
type T is integer range  0..100;
for T'SIZE use 32;
subtype T2 is T range 0..10;
```

F.4 REPRESENTATION ATTRIBUTES

The Representation attributes listed below are as described in the *Reference Manual for the Ada Programming Language*, Section 13.7.2.

X'ADDRESS

NOTE

Attribute ADDRESS is not supported for labels.

X'SIZE

R.C'POSITION

R.C'FIRST_BIT

R.C'LAST_BIT

T'STORAGE_SIZE	for access types, returns the current amount of storage reserved for the type. If a T'STORAGE_SIZE representation clause has been specified, then the amount specified is returned; otherwise, the current amount allocated is returned.
T'STORAGE_SIZE	for task types or objects is not implemented. It returns 0.

F.4.1 Representation Attributes of Real Types

P'DIGITS — yields the number of decimal digits for the subtype P. This value is six for type FLOAT, and 15 for type LONG_FLOAT.

P'MANTISSA — yields the number of binary digits in the mantissa of P. The value is 21 for type FLOAT, and 51 for type LONG_FLOAT.

DIGITS	MANTISSA	DIGITS	MANTISSA	DIGITS	MANTISSA
1	5	6	21	11	38
2	8	7	25	12	41
3	11	8	28	13	45
4	15	9	31	14	48
5	18	10	35	15	51

P'EMAX — yields the largest exponent value of model numbers for the subtype P. The value is 84 for type FLOAT, and 204 for type LONG_FLOAT.

DIGITS	EMAX	DIGITS	EMAX	DIGITS	EMAX
1	20	6	84	11	152
2	32	7	100	12	164
3	44	8	112	13	180
4	60	9	124	14	192
5	72	10	140	15	204

P'EPSILON — yields the absolute value of the difference between the model number 1.0 and the next model number above for the subtype P. The value is 16#0.00001# for type FLOAT, and 16#0.0000_0000_0000_4# for type LONG_FLOAT.

DIGITS	EPSILON	DIGITS	EPSILON	DIGITS	EPSILON
1	16#0.1#E00	6	16#0.1#E-4	11	16#0.8#E-9
2	16#0.2#E-1	7	16#0.1#E-5	12	16#0.1#E-9
3	16#0.4#E-2	8	16#0.2#E-6	13	16#0.1#E-10
4	16#0.4#E-3	9	16#0.4#E-7	14	16#0.2#E-11
5	16#0.8#E-4	10	16#0.4#E-8	15	16#0.4#E-12

P'SMALL — yields the smallest positive model number of the subtype P. The value is 16#0.8#E-21 for type FLOAT, and 16#0.8#E-51 for type LONG_FLOAT.

VALUES	SMALL	VALUES	SMALL	VALUES	SMALL
1	16#0.8#E-5	6	16#0.8#E-21	11	16#0.8#E-38
2	16#0.8#E-8	7	16#0.8#E-25	12	16#0.8#E-41
3	16#0.8#E-11	8	16#0.8#E-28	13	16#0.8#E-45
4	16#0.8#E-15	9	16#0.8#E-31	14	16#0.8#E-48
5	16#0.8#E-18	10	16#0.8#E-35	15	16#0.8#E-51

P'LARGE — yields the largest positive model number of the subtype P. The value is 16#0.FFFFFF8#E21 for type FLOAT, and 16#0.FFFF_FFFF_FFFF_E#E51 for type LONG_FLOAT.

VALUES	LARGE
1	16#0.F8#E5
2	16#0.FF#E8
3	16#0.FFE#E11
4	16#0.FFFE#E15
5	16#0.FFFF_C#E18
6	16#0.FFFF_F8#E21
7	16#0.FFFF_FF8#E25
8	16#0.FFFF_FFF#E28
9	16#0.FFFF_FFFE#E31
10	16#0.FFFF_FFFF_E#E35
11	16#0.FFFF_FFFF_FC#E38
12	16#0.FFFF_FFFF_FF8#E41
13	16#0.FFFF_FFFF_FFF8#E45
14	16#0.FFFF_FFFF_FFFF#E48
15	16#0.FFFF_FFFF_FFFF_E#E51

P'SAFE_EMAX — yields the largest exponent value of safe numbers of type P. The value is 252 for types FLOAT and LONG_FLOAT.

P'SAFE_SMALL — yields the smallest positive safe number of type P. The value is 16#0.8#E-63 for types FLOAT and LONG_FLOAT.

P'SAFE_LARGE — yields the largest positive safe number of the type P. The value is 16#0.FFFF_F8#E63 for type FLOAT, and 16#0.FFFF_FFFF_FFFF_FE#E63 for type LONG_FLOAT.

P'RANGE — yields the range -16#0.FFFF_FF#E63 .. 16#0.FFFF_FF#E63 for type FLOAT, and -16#0.FFFF_FFFF_FFFF_FF#E63 .. 16#0.FFFF_FFFF_FFFF_FF#E63 for type LONG_FLOAT.

P'MACHINE_ROUNDS — is true.

P'MACHINE_OVERFLOWS — is true.

P'MACHINE_RADIX — is 16.

P'MACHINE_MANTISSA — is six for types derived from FLOAT; else 14.

P'MACHINE_EMAX — is 63.

P'MACHINE_EMIN — is -64.

F.4.2 Representation Attributes of Fixed Point Types

For any fixed point type T, the representation attributes are:

T'MACHINE_ROUNDS — true

T'MACHINE_OVERFLOWS — true

F.4.3 Enumeration Representation Clauses

The maximum number of elements in an enumeration type is limited by the maximum size of the enumeration image table which cannot be greater than 65535 bytes. The enumeration table size is determined by the following function:

```
generic
   type ENUMERATION_TYPE is (<>);
function ENUMERATION_TABLE_SIZE return NATURAL is
   RESULT : NATURAL :=0;
begin
   for I in ENUMERATION_TYPE 'FIRST..ENUMERATION_TYPE' LAST loop
       RESULT :=RESULT + 2 + I'WIDTH;
   end loop;
   return RESULT;
end ENUMERATION_TABLE_SIZE;
```

F.4.4 Record Representation Clauses

The *Reference Manual for the Ada Programming Language* states that an implementation may generate names that denote implementation-dependent components. This is not present in this release of the C²Ada Compiler. Implementation dependent offset components are created for record components whose size is dynamic or dependent on discriminants. These offset components have no names.

RESTRICTIONS - Floating point types must be fullword-aligned, that is, placed at a storage position that is a multiple of 32.

Record components of a private type cannot be included in a record representation specification.

Record clause alignment can only be 1, 2 or 4.

Component representations for access types must allow for at least 24 bits.

Component representations for scalar types other than for types derived from LONG_FLOAT must not specify more than 32 bits.

F.4.5 Type Duration

Duration'small equals 61.03515625 microseconds or 2^{-14} seconds. This number is the smallest power of two which can still represent the number of seconds in a day in a fullword fixed point number.

System.tick equals 10ms. The actual computer clock-tick is 1.0/120.0 seconds (or about 8.33333ms) in 60HZ areas and 1.0/100.0 seconds (or 10ms) in 50HZ areas. System.tick represents the greater of the actual clock-tick from both areas.

Duration'small is significantly smaller than the actual computer clock-tick. Therefore, the least amount of delay possible is limited by the actual clock-tick. The delay of duration'small follows this formula:

<actual-clock-tick> ± <actual-clock-tick> + 1.3ms

The 4.45ms represents the overhead or the minimum delay possible on a Model 3280 or 3280MPS Family of Processors. For 60HZ areas, the range of delay is approximately from 1.3ms to 17.97ms. For 50HZ areas, the range of delay is approximately from 1.3ms to 21.3ms. However, on the average, the delay is slightly greater than the actual clock-tick.

In general, the formula for finding the range of a delay value, x, is:

nearest_multiple(x,<actual-clock-tick>) ± <actual-clock-tick> + 1.3ms

where nearest_multiple rounds *x* up to the nearest multiple of the actual clock-tick.

TABLE F-2. TYPE DURATION

DURATION'DELTA	2#1.0#E-14	≃ 61μs
DURATION'SMALL	2#1.0#E-14	≃ 61μs
DURATION'FIRST	-131072.00	≃ 36 hrs
DURATION'LAST	131071.99993896484375	≃ 36 hrs
DURATION'SIZE	32	

F.5 ADDRESS CLAUSES

Address clauses are implemented for objects. No storage is allocated for objects with address clauses by the compiler. The user must guarantee the storage for these by some other means (e.g., through the use of the absolute instruction found in the *Common Assembly Language/32 (CAL/32) Reference Manual*). The exception PROGRAM_ERROR is raised upon reference to the object if the specified address is not in the program's address space or is not properly aligned.

RESTRICTIONS - Address clauses are not implemented for subprograms, packages or task units. In addition, address clauses are not available for use with task entries (i.e.,

interrupts).

Initialization of an object that has an address clause specified is not supported. Objects with address clauses may also be used to map objects into global task common (TCOM) areas. See Chapter 4 for more information regarding task common.

F.6 THE PACKAGE SYSTEM

The package SYSTEM, provided with the C³Ada system permits access to machine-dependent features. The specification of the package SYSTEM declares constant values dependent on the Series 3200 Processors. The following is a listing of the visible section of the package SYSTEM specification.

```
package SYSTEM is

   type ADDRESS is private;

   type NAME is (CCUR_3200);

   SYSTEM_NAME   : constant NAME := CCUR_3200;
   STORAGE_UNIT  : constant := 8;
   MEMORY_SIZE   : constant := 2 ** 24;
   MIN_INT       : constant := - 2_147_483_648;
   MAX_INT       : constant := 2_147_483_647;
   MAX_DIGITS    : constant := 15;
   MAX_MANTISSA  : constant := 31;
   FINE_DELTA    : constant := 2#1.0#E-31;
   TICK          : constant := 0.01;

   type UNSIGNED_SHORT_INTEGER is range 0 .. 65_535;

   type UNSIGNED_TINY_INTEGER is range 0 .. 255;

   type SEMAPHORE_MODE is (NOT_EXECUTED, EXECUTED);

   type SEMAPHORE is private;

   for UNSIGNED_SHORT_INTEGER'SIZE use 16;

   for UNSIGNED_TINY_INTEGER'SIZE use 8;

   subtype PRIORITY      is INTEGER range 0 .. 255;

   subtype BYTE          is UNSIGNED_TINY_INTEGER;

   subtype ADDRESS_RANGE is INTEGER range 0 .. 2 ** 24 - 1;

   ADDRESS_NULL : constant ADDRESS;

   --These functions efficiently copy aligned elements of the specified size.
   --You can declare them locally using any scalar types with
   --PRAGMA interface(assembler,<Routine>);
   --WARNING: these routines work for scalar types only!!!!!!

   function  COPY_DOUBLEWORD (FROM : LONG_FLOAT) return LONG_FLOAT;
   pragma INTERFACE (ASSEMBLER, COPY_DOUBLEWORD);

   function  COPY_FULLWORD (FROM : INTEGER) return ADDRESS;

   function  COPY_FULLWORD (FROM : ADDRESS) return INTEGER;
   pragma INTERFACE (ASSEMBLER, COPY_FULLWORD);

   function  COPY_HALFWORD (FROM : SHORT_INTEGER) return SHORT_INTEGER;
   pragma INTERFACE (ASSEMBLER, COPY_HALFWORD);

   function  COPY_BYTE (FROM : TINY_INTEGER) return TINY_INTEGER;
   pragma INTERFACE (ASSEMBLER, COPY_BYTE);

   function  MEMORY_USED return NATURAL;
   pragma INTERFACE (ASSEMBLER, MEMORY_USED);
```

```
   function  HEAP_USED return NATURAL;
   pragma INTERFACE (ASSEMBLER, HEAP_USED);

   --Address conversion routines

   function  INTEGER_TO_ADDRESS (ADDR : ADDRESS_RANGE) return ADDRESS
      renames COPY_FULLWORD;

   function  ADDRESS_TO_INTEGER (ADDR : ADDRESS) return ADDRESS_RANGE
      renames COPY_FULLWORD;

   function  "+" (ADDR   : ADDRESS;
                  OFFSET : INTEGER) return ADDRESS;

   function  "-" (ADDR   : ADDRESS;
                  OFFSET : INTEGER) return ADDRESS;

   procedure EXECUTE_OR_WAIT (S      : in out SEMAPHORE;
                              S_MODE : out SEMAPHORE_MODE);

   procedure COMPLETED_EXECUTION (S : in out SEMAPHORE);

   procedure RESET_SEMAPHORE (S : in out SEMAPHCRE);

   --This is a 32-bit type which is passed by value

   type EXCEPTION_ID is private;

   function  LAST_EXCEPTION_ID return EXCEPTION_ID;

private

   type ADDRESS is access INTEGER;

   ADDRESS_NULL : constant ADDRESS := null;

   type EXCEPTION_ID is new INTEGER;

   type SEMAPHORE is
      record
         SEMA_OBJ : INTEGER := 0;
      end record;
   pragma INTERFACE (ASSEMBLER, EXECUTE_OR_WAIT);
   pragma INTERFACE (ASSEMBLER, COMPLETED_EXECUTION);
   pragma INTERFACE (ASSEMBLER, RESET_SEMAPHORE);

end SYSTEM;
```

F.7 INTERFACE TO OTHER LANGUAGES

Pragma INTERFACE is implemented for two languages, ASSEMBLER and FORTRAN. The pragma can take one of three forms:

1. For any assembly language procedure or function:

```
pragma INTERFACE (ASSEMBLER, ROUTINE_NAME);
```

2. For FORTRAN functions with only in parameters or procedures:

```
pragma INTERFACE (FORTRAN, ROUTINE_NAME);
```

3. For FORTRAN functions that have in out or out parameters:

```
pragma INTERFACE (FORTRAN, ROUTINE_NAME, IS_FUNCTION);
```

In the C³Ada system, functions cannot have in out or out parameters so the Ada specification for the function is written as a procedure with the first argument being the function return result. Then, the parameter IS_FUNCTION is specified to inform the compiler that it is, in reality, a FORTRAN function. Interface routine_names are truncated to an 8 character maximum length.

F.8 INPUT/OUTPUT (I/O) PACKAGES

The following two system-dependent parameters are used for the control of external files:

- NAME parameter
- FORM parameter

The NAME parameter must be an OS/32 file name string. Figure F-1 illustrates the four fields in an OS/32 file descriptor.

424-3-1

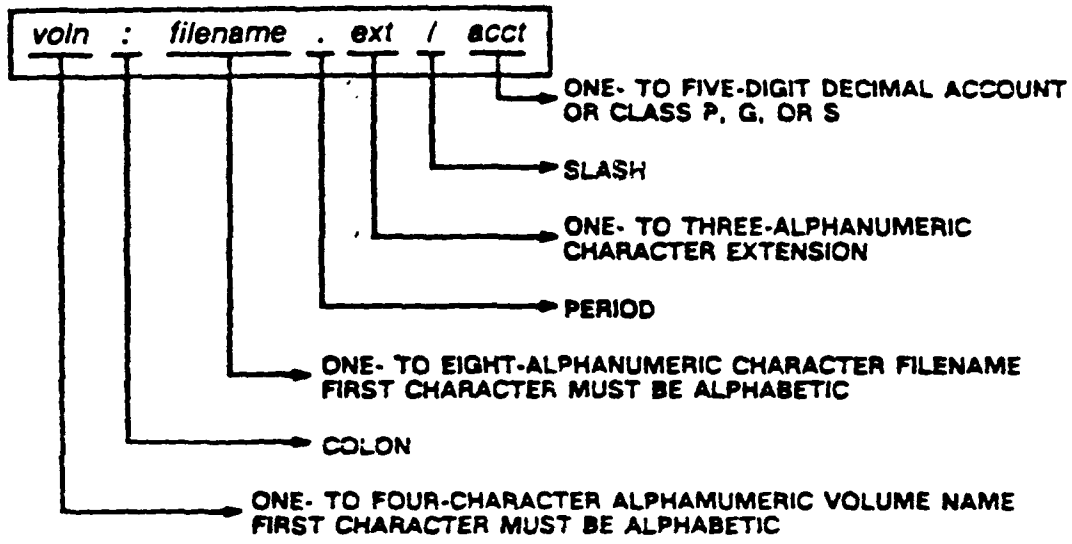


Figure F-1. OS/32 File Descriptor

The implementation-dependent values used for keywords in the FORM parameter are discussed below. The FORM parameter is a string that contains further system-dependent characteristics and attributes of an external file. The FORM parameter is able to convey to the file system information on the intended use of the associated external file. This parameter is used as one of the specifications for the CREATE procedure and the OPEN procedure. It specifies a number of system-dependent characteristics such as fu, file format, etc. It is returned by the FORM function.

The syntax of the FORM string, in our implementation, uses Ada syntax conventions and is as follows:

```
form_param   ::= [form_spec {, form_spec}]
form_spec    ::= lu_spec | fo_spec |
                 rs_spec | dbf_spec |
                 ibf_spec | al_spec |
                 pr_spec | keys_spec |
                 pad_spec | dc_spec |
                 da_spec | ds_spec |
                 ps_spec | ch_spec
lu_spec      ::= LU => lu
fo_spec      ::= FILE_ORGANIZATION => fo
rs_spec      ::= RECORD_SIZE => rs
dbf_spec     ::= DATA_BLOCKING_FACTOR => dbf
ibf_spec     ::= INDEX_BLOCKING_FACTOR => ibf
al_spec      ::= ALLOCATION => al
pr_spec      ::= PRIVILEGE => pr
keys_spec    ::= KEYS => keys
pad_spec     ::= PAD => pad
dc_spec      ::= DEVICE_CODE => dc
da_spec      ::= DEVICE_ATTRIBUTE => da
ds_spec      ::= DEVICE_STATUS => ds
ps_spec      ::= PROMPTING_STRING => ps
ch_spec      ::= CHARACTER_IO
```

The exception USE_ERROR is raised if a given FORM parameter string does not have the correct syntax or if certain conditions concerning the OPEN or CREATE statements are not fulfilled. Keywords that are listed above in upper-case letters are also recognized by the compiler in lower-case.

lu — an integer in the range 0..254 specifying the logical unit (lu) number.

fo — specifies legal OS/32 file formats (file organization). They are:

INDEX | IN
CONTIGUOUS | CO
NON_BUFFERED | NB
EXTENDABLE_CONTIGUOUS | EXTENDABLE_CONTIGUOUS | EC
LONG_RECORD | LR
ITAM
DEVICE

rs — an integer in the range 1..65535 specifying the physical record size.

1. For INDEX, ITAM (inter telecommunications access method) and NON_BUFFERED files, this specifies the physical record size.
2. The physical record size for CONTIGUOUS and EXTENDABLE_CONTIGUOUS files is determined by rounding the element size up to the nearest 256-byte boundary. For such files, *rs* is ignored.
3. The physical record size for LONG_RECORD files is specified by the data blocking factor multiplied by 256 and *rs* is ignored.
4. For a DEVICE the physical record size always equals the element size and *rs* is ignored.

dbf — Data_blocking_factor. An integer in the range 0..255 (as set up at OS/32 system generation (sysgen) time) that specifies the number of contiguous disk sectors (256 bytes) in a data block. It applies only to INDEX, NON_BUFFERED, EXTENDABLE_CONTIGUOUS and LONG_RECORD files. For other file organizations (see *file_organization* above), it is ignored. A value of 0 causes the data blocking factor to be set to the current OS/32 default.

ibf — Index_blocking_factor. An integer in the range 0..255 (as set up at OS/32 sysgen time) specifying the number of contiguous disk sectors (256 bytes) in an index block of an INDEX, NON_BUFFERED, EXTENDABLE_CONTIGUOUS or LONG_RECORD file. For other file organizations (see *file_organization* above), it is ignored.

al Allocation. An integer in the range 1..2,147,483,647. For CONTIGUOUS files, it specifies the number of 256 byte sectors. For ITAM files, it specifies the physical block size in bytes associated with the buffered terminal. For other file organizations, (see *file_organization* above), it is ignored.

pr Privileges. Specifies OS/32 access privileges, e.g., shared read-only (SRO), exclusive read-only (ERO), shared write-only (SWO), exclusive write-only (EWO), shared read/write (SRW), shared read/exclusive write (SREW), exclusive read/shared write (ERSW) and exclusive read/write (ERW).

keys READ/WRITE keys. A decimal or hexadecimal integer specifying the OS/32 READ/WRITE keys, which range from *16#0000# to 16#FFFF#(0..65535).* The left two hexadecimal digits signify the write protection key and the right two hexadecimal digits signify the read protection key. For more information on protection keys, see the *OS/32 Multi-Terminal Monitor (MTM) Primer.*

pad Pad character. Specifies the padding character used for READ and WRITE operations; the pad character is either NONE, BLANK or NUL. The default is NONE.

TABLE F-3. PAD CHARACTER OPTIONS

PAD CHARACTER	ACTION
NONE	Records are not padded. (Default.)
NUL	Records are padded with ASCII.NUL.
BLANK	Records are padded with blanks and OS/32 ASCII I/O operations are used.

dc Device code. An integer in the range 0..255 specifying the OS/32 device code of the external file. See the *System Generation/32 (SYSGEN/32) Reference Manual* for a list of all devices and their respective codes.

da Device attributes. An integer in the range 0..65535 specifying the OS/32 device attributes of the external file. See the *OS/32 Supervisor Call (SVC) Reference Manual* (Chapter 7, the table entitled Description and Mask Values of the Device Attributes Field) for all devices and their respective attributes.

ds Device status. An integer in the range 0..65535 specifying the status of the external file. A status of 0 means that the access to the file terminated with no errors; otherwise a device error has occurred. For errors occurring during READ and WRITE operations, the status values and their meanings are found in Chapter 2 (The tables on Device-Independent and Device-Dependent Status Codes) of the *OS/32 Supervisor Call (SVC) Reference Manual.*

ps Prompting string. This quoted string is output on the terminal before the GET operation only if the file is associated with a terminal; otherwise this FORM parameter is ignored. The default is the null string, in which case no string is output to the terminal.

character_io If character_io is specified in the FORM string, the only other allowable FORM parameters are LU => *lu*, FILE_ORGANIZATION => DEVICE and PRIVILEGE=> SRW. Furthermore, the NAME string must denote a terminal or interactive device. In order for character_io to work properly, the user must specify ENABLE TYPEAHEAD to MTM, to turn on BIOC's type ahead feature.

F.8.1 *Text Input/Ouput (I/O)*

There are two implementation-dependent types for TEXT_IO: COUNT and FIELD. Their declarations implemented for the C³Ada Compiler are as follows:

```
type COUNT is range 0 ..INTEGER'LAST;
subtype FIELD is INTEGER range 0 ..255;
```

F.8.1.1 End of File Markers

When working with text files, the following representations are used for end of file markers. A line terminator followed by a page terminator is represented by:

```
ASCII.FF  ASCII.CR
```

A line terminator followed by a page terminator, which is then followed by a file terminator is represented by:

```
ASCII.FF ASCII.EOT ASCII.CR
```

End of file may also be represented as the physical end of file. For input from a terminal, the combination above is represented by the control characters:

```
ASCII.FF ASCII.EOT ASCII.CR
```

or with BIOC:

```
ASCII.DC4 ASCII.EOT ASCII.CR, i.e., ^T ^D <cr>
```

F.8.2 Restrictions on ELEMENT_TYPE

The following are the restrictions concerning ELEMENT_TYPE:

1. I/O of access types is undefined, although allowable; i.e., the fundamental association between the access variable and its accessed type is ignored.
2. The maximum size of a variant data type is always used.
3. If the size of the element type is exceeded by the physical record length, then during a READ operation the extra data on the physical record is lost. The exception DATA_ERROR is not raised.
4. If the size of the element type exceeds the physical record length during a WRITE operation, the extra data in the element is not transferred to the external file and DATA_ERROR is not raised.
5. I/O operations on composite types containing dynamic array components will not transfer these components because they are not physically contained within the record itself.

F.8.3 TEXT Input/Output (I/O) on a Terminal

A line terminator is detected when either an ASCII.CR is input or output, or when the operating system detects a full buffer. No spanned records with ASCII.NUL are output.

A line terminator followed by a page terminator may be represented as:

```
ASCII.CR
ASCII.FF ASCII.CR
```

if they are issued separately by the user, e.g., NEW_LINE followed by a NEW_PAGE. The same reasoning applies for a line terminator followed by a page terminator, which is then followed by a file terminator.

All text I/O operations are buffered, unless for CHARACTER_IO is specified. This means that physical I/O operations are performed on a line by line basis, as opposed to a character by character basis. For example:

```
put ("Enter Data");
get_line (data, len);
```

will not output the string "Enter Data" until the next put_line or new_line operation is performed.

F.9 UNCHECKED PROGRAMMING

Unchecked programming gives the programmer the ability to circumvent some of the strong typing and elaboration rules of the Ada language. As such, it is the programmer's responsibility to ensure that the guidelines provided in the following sections are followed.

F.9.1 Unchecked Storage Deallocation

The unchecked storage deallocation generic procedure explicitly deallocates the space for a dynamically acquired object.

Restrictions:

This procedure frees storage only if:

1. The object being deallocated was the last one allocated of all objects in a given declarative part.
2. All objects in a single chunk of the collection belonging to all access types declared in the same declarative part are deallocated.

F.9.2 Unchecked Type Conversions

The unchecked type conversion generic function permits the user to convert, without type checking, from one type to another. It is the user's responsibility to guarantee that such a conversion preserves the properties of the target type.

Restrictions:

The object used as the parameter in the function may not have components which contain dynamic or unconstrained array types.

If the target'size is greater than the source'size, the resulting conversion is unpredictable. If the target'size is less than the source'size, the result is that the left-most bits of the source are placed in the target.

Since unchecked_conversion is implemented as an arbitrary block move, no alignment constraints are necessary on the source or the target operands.

F.10 IMPLEMENTATION-DEPENDENT RESTRICTIONS

1. The main procedure must be parameterless.
2. The source line length must be less than or equal to 80 characters.
3. Due to the source line length, the largest identifier is 80 characters.
4. No more than 65534 lines in a single source file.
5. The maximum number of library units is 9999.
6. The maximum number of bits in an object is $2^{31}-1$.
7. The maximum static nesting level is 63.
8. The maximum number of directly imported units of a single compilation unit must not exceed 255.
9. Recompilation of SYSTEM or CALENDAR specification is prohibited.
10. ENTRY'ADDRESS, PACKAGE'ADDRESS and LABEL'ADDRESS are not supported.
11. The maximum number of nested SEPARATES is 63.
12. The maximum length of a filename is 80 characters.
13. The maximum length of a program library name is 64 characters.
14. The maximum length of a listing line is 125 characters.
15. The maximum number of errors handled is 1000.
16. The maximum subprogram nesting level is 64.
17. The maximum number of calls to pragma ELABORATE per compilation unit is 255.
18. The maximum number of unique symbols (identifiers, numeric literals, characters, and strings) per compilation is 12503. This limit includes imported symbols.

19. The total size for text of unique symbols (including imported symbols) per compilation is 100000.
20. The maximum parser stack depth is 10000.
21. The maximum depth of packages is 100.
22. The static aggregate nesting limit is 256.

F.11 UNCONSTRAINED RECORD REPRESENTATIONS

Objects of an unconstrained record type with array components based on the discriminant are allocated using the discriminant value supplied in the object declaration. If the size of an unconstrained component has the potential of exceeding 2 Gb, the exception NUMERIC_ERROR is raised. Assignment of a default maximum discriminant value does not occur. For example:

```
type DYNAMIC_STRING( LENGTH : NATURAL := 10 )
     is record
     STR : STRING( 1 .. LENGTH );
     end record;
DSTR : DYNAMIC_STRING;
```

For this record, the compiler attempts to allocate NATURAL'LAST bytes for the record. Because this is greater than 2GB, the exception NUMERIC_ERROR is raised. However, the declaration

```
D : DYNAMIC_STRING(80);
```

raises no exception and creates a record containing an 80 byte string.

F.12 TASKING IMPLEMENTATION

The C³Ada tasking implementation uses the co-routine paradigm. That is, a task never yields control of the processor until either:

- it is suspended at an accept statement,
- it is suspended at an entry call statement,
- it is suspended at a delay statement (with simple_expression > 0.0),
- an open delay alternative is selected,
- an open terminate alternative is selected,
- it has completed its execution,
- it activates another task,
- it aborts a task, or
- a master construct is suspended while dependent tasks terminate.

There is only one OS/32 task for all Ada tasks in this model.

Tasks that depend on library packages continue to execute when the main program terminates.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..79 => 'A', 80 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..79 => 'A', 80 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..39 => 'A', 40 => '3', 41..80 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..39 => 'A', 40 => '4', 41..80 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..77 => '0', 78..80 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..75 => '0', 76..80 => "690.0")
$BIG_STRING1 A string literal which when catenated with $BIG_STRING2 yields the image of $BIG_ID1.	(1 => '"', 2..41 => 'A', 42 => '"')
$BIG_STRING2 A string literal which when catenated to the end of $BIG_STRING1 yields the image of $BIG_ID1.	(1 => '"', 2..40 => 'A', 41 => '1', 42 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..60 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2**24
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	CCUR_3200
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_SHORT_SHORT_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	4294967295.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	255
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	ILLEGAL_.FIL
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	ILLEGALFILE.NAM
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2147483648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-4294967296.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	0
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	80
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..2 => "2:", 3..77 => '0', 78..80 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..3 => "16:", 4..76 => '0', 77..80 => "F.E:")
$MAX_STRING_LITERAL A string literal of size $MAX_IN_LEN, including the quote characters.	(1 => '"', 2..79 => 'A', 80 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	TINY_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	CCUR_3200
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2**24

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	CCUR_3200
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	0.01

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These

tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY CONCURRENT

Compiler: C^3Ada

ACVC Version: 1.10

Default Values for Compiler Start Options

The following are the Compiler Start Options along with the default values.

- *ABORT => OFF*

  If set, the compiler stops compiling when an error is encountered in a unit and will not compile any subsequent units in the source file.

- *ALIST => OFF*

  If set, a symbolic assembly listing will be appended to the listing file.

- *INFORM => ON*

  If set, all the information messages (if applicable) will appear in the listing file.

- *INLINE => ON*

  If set, all the *pragma Inline* requests will be honored (when applicable).

- *LIST => ON*

  If set, a listing file will be produced.

- *OPTIMIZE => ON*

  If set, the compiler performs some optimizations to improve run-time efficiency of the program.

- *PAGE_SIZE => 60*

  The default specifies that there will be 60 lines per page in the listing file.

- *SEGMENTED => ON*

  If set, the object code will contain both PURE (read only) and IMPURE (read and write) segments.

- *STACK_CHECK => ON*

  If set, the compiler generates extra code to check if there is enough stack space.

- *SUMMARY => ON*

  If set, the compiler generates a summary file when *adacomp* is used with a *$file_list_name*.

- *SUPPRESS_ALL => OFF*

  If set, the compiler will not generate any run-time checking code for all types and objects in the compilation: Access_check, Discriminant_check, Index_check, Length_check, Overflow_check, and Range_check.

- *SUPPRESS_OVERFLOW => OFF*

  If set, the Overflow checking code will not be generated by the compiler.

- *WARN => ON*

  If set, all the warning messages will appear in the listing file.